Techniques for Integrating Computer Programs into Design Architectures

Mark A. Hale* and James I. Craig†

Georgia Institute of Technology
Atlanta, Georgia 30332-0150

Abstract

The benefits of using modular computer architectures for multi-disciplinary design are being explored by industry, government, and academia. These architectures are being validated through a considerable number of in-house and team demonstration projects. Based on experiences to date, a generic computing design architecture consists of the following components: process management, a common product data model, an analysis toolkit, a problem-independent computing backplane, and integration mechanisms. The latter is concerned with the addition of services to computer resources in an analysis toolkit, called *wrapping*, and is discussed in this paper. Wrapping allows for the collaborative use of resources within a computer architecture. Strategies and consequences of integrating resources from executables to source code are outlined. Benefits associated with using software agents to assist designers in integrating and using software resources in design computing architectures are highlighted.

Background

There is ongoing research and developments in computer architecture support multi-disciplinary design activities.[1-9] Based on these architectures, a generic computing design architecture consists of the following components: process management, information management, an analysis toolkit, a problem-independent computing backplane, and integration mechanisms. An example architecture that incorporates this functionality has been developed at Georgia Tech and is called IMAGE (Intelligent Multidisciplinary Aircraft Generation Environment) and is shown in Figure 1.

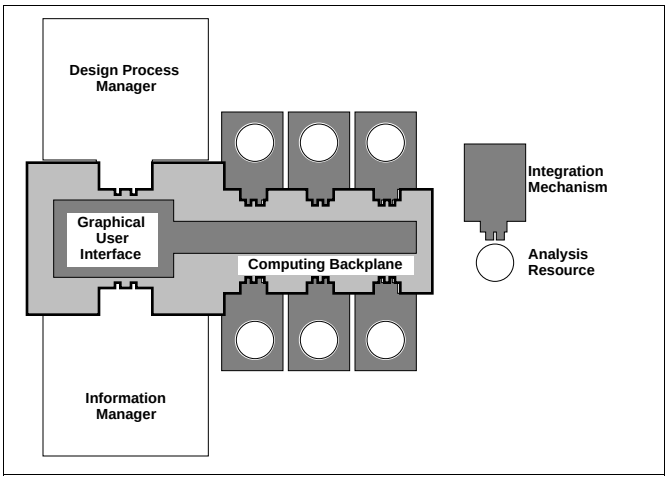


Figure 1. IMAGE Architecture

During design processes, analyses are executed so that the resulting data coalesced for further review and analysis. The addition of computer utilities so that these resources can exchange data *bi-directionally* is called "wrapping". Proper wrapping techniques allow for legacy and new computer resources to be used in modular design architectures such as the one shown in Figure 1. A sample of techniques that can be used for integrating design resources are the focus of this paper. In addition, the benefits of doing so are discussed.

Computer Resources

Computer resources are developed, or a computer architecture designer may choose to integrate them, in one or more forms. These include:

* Graduate Researcher, Aerospace Systems Design Laboratory, Student Member AIAA

mark.hale@cad.gatech.edu
http://www.cad.gatech.edu/image

† Professor, Aerospace Systems Design Laboratory, Member AIAA

Sixth AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Bellevue, WA, September 4-6, 1996

AIAA-96-4166

- *executable,*
- *object module,*
- *source code,*
- equations,
- knowledge bases, and
- scripts.

The resources forms that are italicized are specifically addressed in this paper. Notice that expert systems, incorporated as knowledge bases, are included in this list. These are becoming more popular as heuristics are introduced into design software. This allows for manufacturing and economic considerations to be modeled and integrated with design.

A designer may choose to integrate his resource in one or more of these possible forms. The selection of which form to use for the analysis resources is non-trivial and depends on:

- code availability,
- preservation of proprietary boundaries,
- security interests,
- level of effort desired for resource integration,
- cost expenditure,
- time expenditure,
- resource efficiency,
- validation of existing code, and
- designer preference.

A successful wrapping strategy must assist a design in each of these cases. Moreover, integration should be language and platform independent in order to provide the largest domain of resource support.

Some current architecture implementations face considerable limitations and opposition because of the degree of source code modification required for resource integration. A suite of integration techniques is outlined in this paper that minimize and/or eliminate the need to do modifications. These techniques are intended for use on UNIX ™ based systems and are extendible to other platforms. These techniques specifically address the integration of executables, object modules, and source code forms of analysis resources into a design framework.

Generic Wrap

The combination of a resource and a wrap is called a software *tool*. This is shown in Figure 2. Additional services can be added to make these tools into software *agents*. The primary difference between a tool and an agent is the addition of a model in an agent. A model captures the behavioral characteristics of the resource and thus allows for accountable resource use. This has been shown to be a very powerful asset in design systems and is discussed in the closing sections of this paper.

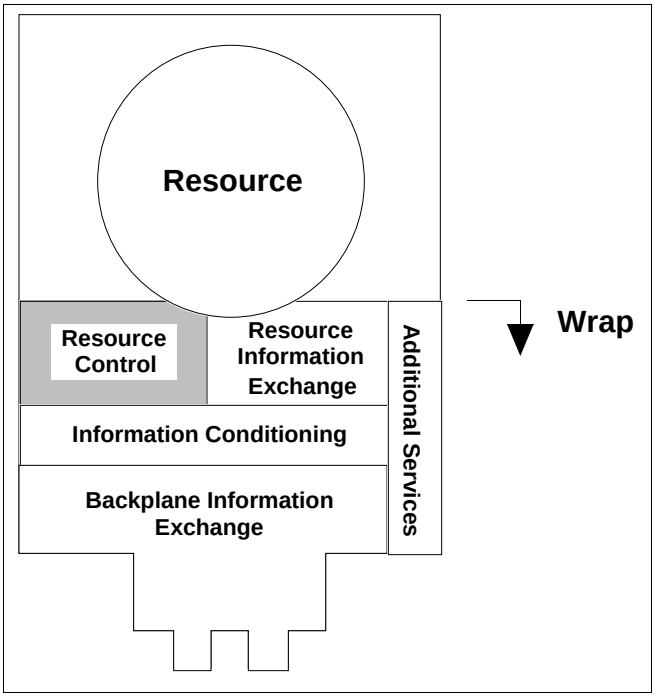


Figure 2. Tool Components

The components required to integrate a resource are shown in Figure 2. A designer uses an architecture which controls and exchanges information with the resource via the wrap. This information must be conditioned to insure compatibility and so that additional services can be provided. These services include model processing, scripting, and publication. In turn, the wrap is responsible for controlling and exchanging information with resource. As shown in Figure 2, the net result is that the resource is buffered from the architecture so that the architecture is not dependent on the particular form of the resource (e.g., executable, object, source).

Tk/tcl

Tk/tcl has been used to demonstrate the wrapping of analysis tools. Tk/tcl is an interpretive windowing system developed at U.C. Berkeley.[10] The features of Tk/tcl make the software an appropriate research vehicle for demonstrating technologies required for software architectures. The **T**ool **C**ommand **L**anguage (tcl) is an interpretive shell similar to the Bourne (ksh) and C (csh) shells used on most UNIX systems. Example shells are shown in Figure 3. Notice that the tcl shell is started from the ksh. The tcl shell permits the use of variables, run-time procedure declaration, and access to compiled

procedures. In addition, UNIX commands can be used via system and exec calls.

```
ksh>set NUMBER=0
ksh>while [ "$NUMBER" -lt 5 ]
do
echo "NUMBER = $NUMBER"
NUMBER=`expr $NUMBER + 1 `
done
NUMBER = 0
NUMBER = 1
NUMBER = 2
NUMBER = 3
NUMBER = 4
ksh>
```

ksh

```
ksh>tcl
tcl>set NUMBER 0
tcl>while { $NUMBER < 5 } {
echo "$NUMBER = $NUMBER"
incr NUMBER 1
}
NUMBER = 0
NUMBER = 1
NUMBER = 2
NUMBER = 3
NUMBER = 4
tcl>
```

tcl

Figure 3. ksh and tcl shells

Tk is a **T**ool**k**it that provides a widget library for developing graphical user interfaces. Widgets include such things as buttons, menus, entry fields, listboxes, text areas, and canvases. Tk is intimately tied to tcl so that they share a common command structure and tcl commands can be bound to widget events. The commands required to draw a button widget are shown in Figure 4.


```
ksh>wish
wish>button .button1 -text "Push Here
to Exit" -command exit -foreground red
-background white
.button1
wish>pack .button1
wish>bind .button1 <Enter> "exit"
wish>
```

Figure 4. Tk Button Widget

The program that combines both Tk and tcl is called "wish". A functional diagram of this program is shown in Figure 5. Commands and widgets are accessed by a user through an interpreter. A standard set of commands and widgets are available through a library of compiled procedures. The interpreter can also be extended by addition additional procedures or libraries.

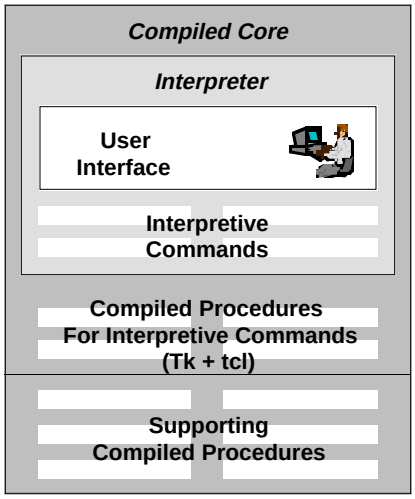


Figure 5. Wish Interpreter

Tk/tcl is a good tool to be used for demonstrating design frameworks because it can be modified and extended easily. New procedures (written in tcl) can be declared or added to the interpreter (written in a language compatible with C functions) Using these facilities, tcl can be used to write the shell scripts that some integrated architectures employ. In addition, tcl can be extended to provide the utilities needed to implement the wrapping strategies described in this paper. For example, the IMAGE architecture developed by the authors is built around the Tk/tcl interpretive windowing system with extensions added for drag & drop (blt), object-oriented data models (itcl), natural language processing (marpa), and message-passing (PVM).[11-13]

Some Wrapping Techniques

When a resource is wrapped, the wrap must have control over the resource as well as a means of exchanging information with the resource. The ability, or perhaps the art, of controlling resources and providing a communications channel with a resource will be specifically addressed in this paper. This portion of wrapping has the greatest dependency on the form of the resource (e.g., executable, object module, source). The methods described here are intended to be a sample of techniques that are available.

Executable

Executables developed as independent applications represent a significant portion of the available engineering software base. These applications are written in any number of languages, operate on a variety of platforms, and have equally as many user and data interfaces. The applications may be developed by individual disciplinarians, integrated product design teams, or corporations. Because of the different implementations and architecture restrictions imposed by using executables, the wrapping of these resources is inherently course and the data streams often require conditioning (parsing, substitutions, etc.). However, the ability to utilize these resources within a computer architecture outweigh the limitations imposed by them. Moreover, the preservation of proprietary data and source boundaries that is possible with executables makes their use worthwhile especially in multi-corporate teaming.

The wrapping of executables depends primarily on the type of user interface. As shown in Table 1, the interface may be graphical or non-graphical. File-based programs are typically executed on the UNIX command line as:

```
program.exe FILE.IN FILE.OUT
```

or may use default files. Programs that require user-input may often be converted into file-based programs by using re-direction, for example:

```
program.exe < FILE.IN > FILE.OUT
```

Fortunately, a majority of legacy design analysis software falls under one of these two categories.

Table 1. Executable Wrapping Techniques

Non-Graphical		Graphical
File-based	User input	
system call	exec call	event handling
pipe	pipe	signals
fork	fork	stream
rsh	expect	manipulation
rcmd		
expect		

A common misconception of design frameworks is that they are fully automated. With this in mind, *an important consideration is whether or not a design resource should be automated or require user interaction.* It can be said that the ability to wrap graphical applications to support automated data transfer is considerably more difficult than batch mode applications because of the need to provide event handling. Graphical applications are best wrapped by continuing to allow a designer to interact with the resource during data transfer for file creation. Instead of simulating a user's events, the wrap would guide an end-user through a proper sequence of events.

Another concern about the use of exectables is the amount of effort required to integrate them. A significant portion of the integration effort is spent in determing variable names and bounds, file structures, and different analysis capabilities. This effort is required to be expended regardless of the software architecture that is chosen. However, tools can be developed that facilitate these processes and keep the underlying computer architecture transparent.

The wrap must provide access to the executable, which is an independent application. Considering Tk/tcl as a wrapper, an executable would be functionally located externally to the interpreter as shown in Figure 6, and connected via one of the threads outlined in Table 1. Notice that there are no inter-dependencies between the wrap and the executable. [14]

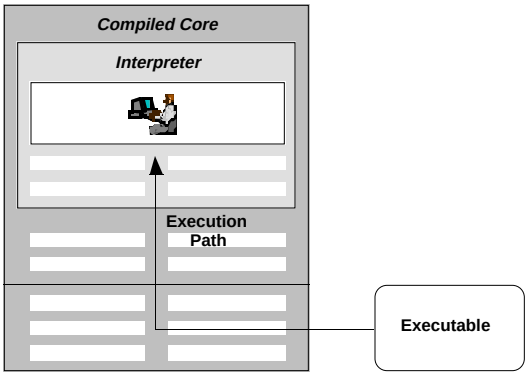


Figure 6. Executable-Interpreter Relationship

Object Module

More flexibility can be exercised when using object modules versus executables because subroutine control and argument passing reside with the wrap. In addition to other things, this flexibility can be used to provide better data management, improved process descriptions, and parallel analyses. This flexibility is based on the assumptions that:

- the object modules represent analysis routines of interest;
- inter-dependency of the modules are sufficiently small;
- there are no conflicts (variables, functions, etc.) with the wrapper; and
- the modules are not main entry points.

It should be pointed out that these types of conflicts prohibit some codes from being wrapped without modifications to the source code. In addition, the use of an object module preserves some of the data and source boundaries that executables have. However, the control and function arguments are no longer masked.

It is possible to convert object modules into executables and use the methods described in the previous section for wrapping. Techniques for wrapping object modules are listed in Table 2. Notice that these techniques only describe the controlling interface and the file and input issues described in the previous section still need to be addressed. The capability to link object modules with the wrap must be insured as the wrapping tools is being designed or selected.

Table 2. Object Module Wrapping Techniques

compilation into the architecture	dynamic method calls
signal	cross-language support
sockets	shared memory
remote procedure calls	

In relation to Tk/tcl, an object can be compiled into a separate executable and have a configuration as shown in Figure 6. Or, the object can be imbedded within the compiled core and linked to the interpreter, see Figure 7. A new command is provided in the interpreter so that the procedure can be accessed. The geometry resources, consisting of FORTRAN subroutines, provided by CATIA™ (a 3D solids modeling system) have been wrapped in this manner and is documented in Reference [15]. Using this wrap, a designer can create solids models in CATIA automatically without having to write and compile FORTRAN code.

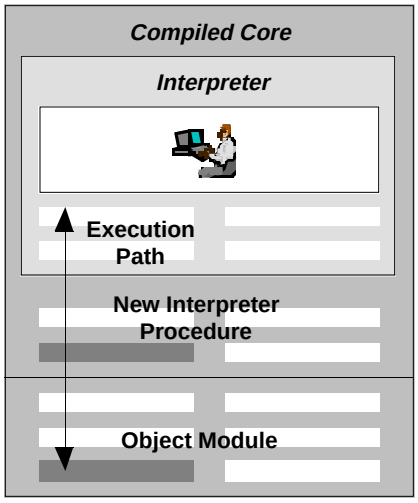


Figure 7. Object Module-Interpreter Relationship

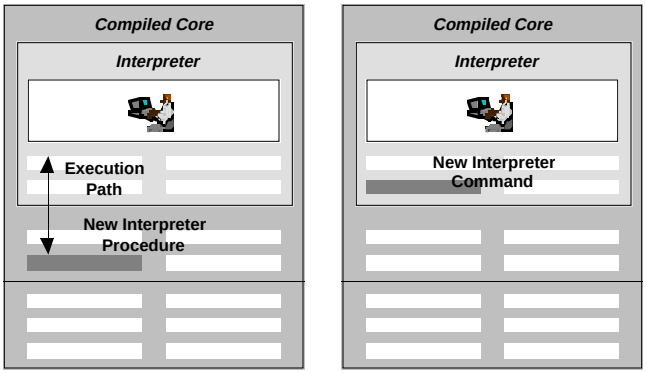
Source Code

By nature, source code provides maximum flexibility but does so without preserving proprietary boundaries. At a minimum, source code would be wrapped by using the code as object modules and employ the methods from the previous section. This code may be modified slightly to make argument passing easier. The benefits of having source code access are realized if the additional services that may be available in a computing architecture are used. These services include:

- Structure passing,
- Data tracking,
- Timing, and
- Signal handling.

Codes linked together using some of these services are referred to as having fine-grained parallelism.

Again, it is possible to convert source code into object modules or executables and use methods previously described, see Figure 6 and Figure 7. If source code is available, the source could be coded directly into a compiled procedure, see Figure 8a, or implemented as an interpretive command, see Figure 8b.



a) Compiled Procedure b) Interpretive Command

Figure 8. Source-Interpreter Integration

Agents

Agents provide the capabilities required to implement a design architecture used for multi-disciplinary design. These include accountability, reconfigurable design processes and data parameters, and a teaming approach to decision-making. To accomplish this functionality, agents extend the tools that have been described so far.

A formal agent definition is given in Reference [15]:

> *An agent is a resource that has been modeled and wrapped for inclusion in a distributed design environment. Agent design requires a designer-centered, bi-directional wrap that is independent of proprietary boundaries and capable of supporting increasing fidelity models.*

The addition of the model is the most noticeable difference between an agent and the tools described earlier. This is shown in Figure 9.

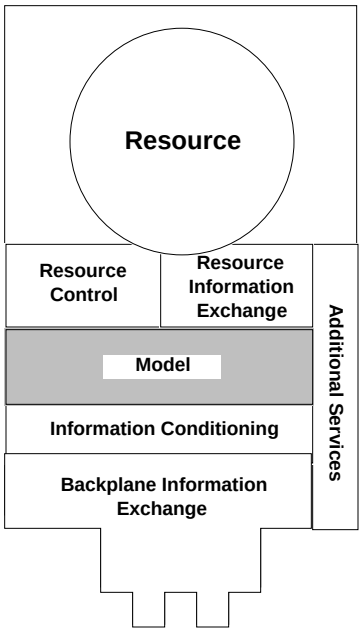


Figure 9. Agent Components

Model

An agent's model has two components:

1) *Process Model*; describes the behavior of an agent
2) *Implementation Model*; describes the implementation of an agent

The Process Model may be physical or intellectual. These are typically based on mathematical formulations, engineering principles, or geometrical constructions. The Process Model has typically been discarded or included only in external references guides The use of Agents allows for Process Models to be explicitly defined. For example, a solids construction model used to represent complex solids in CATIA™ is shown in Figure 10. In words, the geometric Process Model describing the volume transformation would be:

> In a volume transformation, an object is represented by an approximate solid computed directly from the exact volume. A volume is constructed from faces which, in turn, are defined by the edges that enclose simple or multiply connected regions of planar or complex surfaces.

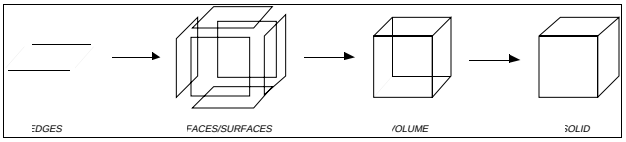


Figure 10. CATIA™ Solid Representation

The Implementation Model, the second model component, captures the execution characteristics of the resource. Some of the items that are contained in the Implementation Model include: variable definition, file descriptions, units, execution characteristics, and platform dependencies.

An agent may have the ability to process multiple models. For instance, a CATIA™ agent may model both the complex solids constructions defined above as well as the more common Boolean solids construction. Moreover, the same agent may utilize multiple resources. On a larger scale, a rendering agent may include CATIA™ as well as PRO-ENGINEER™ as solids modeling software resources.

Though specific modeling languages are still under investigation, a Model can still be used to describe agent operations.[4, 14] One modeling perspective would be an interpretive model much like a Mathematica™ Workbook. Mathematica™ could be used to describe analysis algorithms in terms of symbolic, algebraic expressions. A similar idea was adopted in IMAGE whereby models are contained in the agent and processed symbolically based on an algorithm encoded into Tk/tcl. Though currently the processing algorithm is simple, it nonetheless serves as a means of exploring modeling concepts. For example, the CATIA™ solids example of Figure 10 is processed using the model shown in Figure 11. Tcl commands are italicized.

```
Edges = Create_Cube_Edges Origin \
            Side_Length
Faces = Create_Cube_Faces Edges
Volume = Create_Cube_Volume Faces
Solid = Create_Solid Volume
```

Figure 11. Process Model for a Solid Cube Constructed in CATIA™

Accountability

As software is integrated into larger systems, accountability issues become increasingly more important. Commercial software is often validated and

tested before it goes to market. However, does the validity of these codes remain intact in larger architectures? One may argue that the codes are valid if wrapping techniques preserve proprietary boundaries since the program structure remains intact. Moreover, the incorporation of a wrap within an agent allows for more responsible use of software since software limitations, scope, and assumptions are known. This serves to minimize the "garbage in - garbage out" syndrome.

Design data (information) must also be accountable along with the resources used to generate it. The use of Models and some of the model processing techniques described in the previous section allows for data to automatically be generated in context. Therefore, a designer has the following information available to him:

- the agent used to create the data,
- the data required by the agent,
- the time at which the agent was solicited,
- the resources queued by the agent, and
- who solicited the information.

Context provides the extra design information required for responsible decision making. This is becoming more important as Integrated Product Teams (IPT) perform design-related activities.

Publication

The use of agent allows for design publication for execution, decision-making and review. Wraps have the capability to publish models present within agents, one of the additional service facilities eluded to in Figure 2. Thus, agent capabilities become known in larger architectures. An IPT would be aware of the potential analysis capabilities available to it so that the team can lay out design processes and delegate tasks. In addition, Model publication assists in the brokerage of services within a collaborative environment since agent services are documented in the Process and Implementation models.

In addition, agents allow for their usage to be published. This information can be assembled into a design chronology.[16] Based on this history, a designer or IPT can base decisions on the events that lead up to the current point in the design analysis.

Design Flexibility

Design processes are inherently dynamic. As designs progress and decisions are made, particular processes may be added, changed, or eliminated. The "wires" that establish the connections between software resource, or agents, are dependent on these processes. The flexibility imposed under these conditions exemplifies the need for multi-disciplinary design architectures to remain problem independent. This behavior is exhibited by agents as their models are combined with data schemas and assembled into design processes.

Optimizers are commonly used in vehicle design. A simple aircraft design process is shown in Figure 12 where gradients are determined for an optimization tool. The design process shown in the figure can be changed to accomodate more complex design processes or the incorporation of more sophisticated analysis. For instance, CONMIN, DOT, or KSOPT could each be used as the optimizer in Figure 12. To link one of these optimizers, the variables are matched between those available in the optimizer (X_1, X_2) and the design variables being studied in a current aircraft design ($/RPM, Total Weight). As can be seen from Figure 12, the connections allow for design problems to be configured independent from software or hardware considerations.

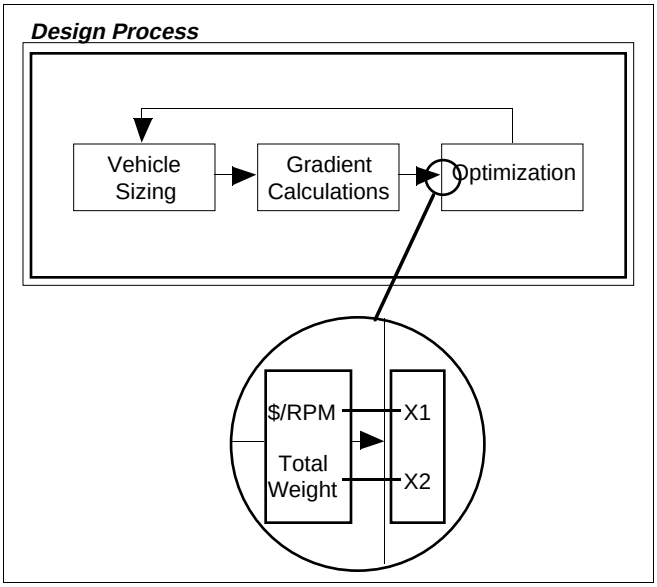


Figure 12. A Configurable Design Process

Conclusion

Wrapping is a method for linking together software resources in design architectures. Wrapping is dependent on the type of resource that is available. Strategies were shown for executables, object modules, and source code. Proprietary boundaries of software can preserved but in some cases at the expense of analysis flexibility. Agents extend the capabilities of these software tools by incorporating software modeling facilities allowing for accountable resource utilization and publication. In addition, these capabilities allow for configurable design processes. The functionality given by wrapping and agents eventually plays a role in providing the implementation of multi-disciplinary design activities.

Acknowledgments

Funding for this paper is provided by the NASA Graduate Student Researchers Program (NGT-51250) under the direction of NASA Langley's High Performance Computing and Communications Program. Software and hardware support is provided by the CAE/CAD Laboratory at the Georgia Institute of Technology.

References

1. Chapman, B., P. Mehrotra, J. V. Rosendale and H. Zima, "A Software Architecture for Multidisciplinary Applications: Integrating Task and Data Parallelism," Institute for Computer Applications in Science and Engineering, March 1994. NASA CR-194896, ICASE 94-18.

2. Cutkosky, M. R., R. S. Engelmore, R. E. Fikes, M. R. Genesereth, T. R. Gruber, W. S. Mark, J. M. Tenenbaum and J. C. Weber, "PACT: An Experiment in Integrating Concurrent Engineering Systems," IEEE Computer, vol. 26, no. , pp. 28-37, January, 1993.

3. Dovi, A. R., G. A. Wrenn, J.-F. M. Barthelemy, P. G. Coen and L. E. Hall, "Multidisciplinary Design Integration System for a Supersonic Transport Aircraft," Fourth AIAA / USAF / NASA / OAI Symposium on Multidisciplinary Analysis and Optimization, Cleveland, OH, September 21-23, 1992. AIAA-92-4841.

4. Finin, T., J. Weber, G. Wiederhold, M. Genesereth, R. Frtzon, D. McKay, J. McGuire, R. Pelavin, S. Shapiro and C. Beck, "Specification of the KQML Agent-Communication Language," The DARPA Knowledge Sharing Initiative External Interfaces Working Group, February, 1994.

5. Frank, G. A., J. B. Clary and B. L. Dove, "Design Automation for Concurrent Engineering," 1st AIAA National TQM Symposium, Denver, CO, November 1-3, 1989. AIAA-89-3207.

6. Hale, M. A. and J. I. Craig, "Use of Agents to Implement an Integrated Computing Environment," Computing in Aerospace 10, AIAA, San Antonio, TX, March 28-30, 1995. AIAA-95-1001-CP.

7. Hughes, D., "Generic Command Center Speeds Systems Design," Aviation Week & Space Technology, pp. 52-53, March 8, 1993.

8. Townsend, J. C., R. P. Weston and T. M. Eidson, "An Overview of the Framework for Interdisciplinary Design Optimization (FIDO) Project," NASA Langley Research Center, July, 1994. NASA TM 109058.

9. Hale, M. A., J. I. Craig, F. Mistree and D. P. Schrage, "On the Development of a Computing Infrastructure that Facilitates IPPD from a Decision-Based Perspective," 1st AIAA Aircraft Engineering, Technology and Operations Congress, Los Angeles, CA, September 19-21, 1995. AIAA-95-3880-CP.

10. Ousterholt, J. K., An Introduction to Tcl and Tk. Reading, MA: Addison-Wesly Publishing Company, Inc. 1993.

11. Beguelin, A. L., J. J. Dongarra, G. A. Geist, W. C. Jiang, R. J. Mancheck, B. K. Moore and V. S. Sunderam, "Parallel Virtual Machine," Version 3.3, University of Tennessee, Oak Ridge National Laboratory, Emory University, June, 1993.

12. Kegler, J., "Marpa - A Parser for Hackers," Version 2.8, April, 19, 1995.

13. McLennan, M. J., "Incr Tcl - Object-Oriented Extensions for Tcl," Version 1.2, AT&T Bell Laboratories, March 25, 1994.

14. Hale, M. A., "An Open Computing Infrastructure that Facilitates Integrated Product and Process Development from a Decision-Based Perspective," Doctoral Dissertation, Georgia Institute of Technology, School of Aerospace Engineering, July, 1996.

15. Hale, M. A. and J. I. Craig, "Preliminary Development of Agent Technologies for a Design Integration Framework," Fifth AIAA / NASA / USAF / ISSMO Symposium on Multidisciplinary Analysis and Optimization, Panama City, Florida, September 7-9, 1994. AIAA-94-4297-CP.

16. Stephens, E., "LEGEND: Laboratory Environment for the Generation, Evaluation, and Navigation of Design," Doctoral Dissertation, Georgia Institute of Technology, School of Aerospace Engineering, September 1993.